The Icecube Data Acquisition Software: Lessons Learned During Distributed, Collaborative, Multi-Disciplined Software Development

K S Beattie[1], C T Day[1], D Glowacki[2], K D Hanson[2], J E Jacobsen[3], C P McParland[1] and S J Patton[1]

[1] Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[2] Department of Physics, University of Wisconsin, Madison, WI 53706, USA
[3] NPX Designs, Inc. Chicago, IL 60615, USA

E-mail: KSBeattie@lbl.gov, CTDay@lbl.gov, dave.glowacki@icecube.wisc.edu, kael.hanson@icecube.wisc.edu, john@mail.npxdesigns.com, CPMcParland@lbl.gov, SJPatton@lbl.gov

Abstract. In this experiential paper we report on lessons learned during the development of the data acquisition software for the IceCube project - specifically, how to effectively address the unique challenges presented by a distributed, collaborative, multi-institutional, multi-disciplined project such as this. While development progress in software projects is often described solely in terms of technical issues, our experience indicates that non- and quasi-technical interactions play a substantial role in the effectiveness of large software development efforts. These include: selection and management of multiple software development methodologies, the effective use of various collaborative communication tools, project management structure and roles, and the impact and apparent importance of these elements when viewed through the differing perspectives of hardware, software, scientific and project office roles. Even in areas clearly technical in nature, success is still influenced by non-technical issues that can escape close attention. In particular we describe our experiences on software requirements specification, development methodologies and communication tools. We make observations on what tools and techniques have and have not been effective in this geographically disperse (including the South Pole) collaboration and offer suggestions on how similarly structured future projects may build upon our experiences.

1. Background

1.1. Project overview

The IceCube project is an international collaboration of scientists and engineers constructing a neutrino observatory in the deep clear ice beneath the South Pole at the US Amundsen-Scott station. Upon completion in 2010, IceCube will consist of approximately 5000 digital optical modules (DOMs) located in a volume of roughly one cubic kilometer of highly transparent ice situated between 1450 and 2540 meters below the surface. These sensors will detect Cherenkov light emitted by charged particles moving as a result of collisions in the ice from high-energy neutrinos penetrating the earth moving upward from the northern hemisphere. After initially collecting data about such events, off-line analysis will determine the directions from which these neutrinos came to us and how much energy each carried. One of the goals of this detector is

to discover the origin of these extremely high-energy cosmic rays and identify the sites in the distant universe where these cosmic rays are produced - in effect creating a neutrino map of the northern hemisphere.

As of 2007, the detector is a little over 25% complete with 22 strings in the ice taking data. This is up from 9 strings last year and 1 string in 2005. For a more detailed description of the detector that was is required here, see [1].

1.2. Detector architecture

The construction of the detector consists, in part, of and array of 80 electronic cables or "strings", each containing 60 Digital Optical Modules (DOM), deployed into deep ice holes. Each DOM communicates what it "sees" to a cluster of systems running our Surface Data Acquisition (DAQ) software. The IceCube project consists of considerably more software than this (Monte Carlo Simulation, Processing and Filtering, Satellite transport, Data Warehousing, Off-line analysis, etc.), but in this paper we focus primarily on the experiences of developing the DAQ software.

Though not a complete description of its functionality, for the purposes of this paper, each DOM consists of a transparent pressure sphere housing a photomultiplier tube, a custom-built integrated circuit main board with a waveform digitizer and LED flasher lights. The main board captures the output of the photomultiplier tube, digitizes it, timestamps it and buffers it for delivery to the surface via the string on which the DOM is attached. The main board comprises an embedded processor whose firmware can be reprogrammed from the surface. DOMs are synchronized via a GPS clock to achieve, at worst, 4 nanosecond timing precision across the entire array. Moreover, calibration runs for neighboring DOMs can be taken using the integrated LEDs.

Cables from all strings converge at the surface into the IceCube Laboratory (ICL), a two-story elevated building housing all the electronic resources needed for data-taking, archiving and filtering data, including the compute cluster running the DAQ software. A team of "winter-over" scientists maintain the systems during the austral winter (from February to October).

1.3. DAQ Architecture

What follows here is but a brief description of the current DAQ architecture. The design's evolution over time is discussed in following sections. While, this description leaves out many details, it should suffice for the purposes of this experiential paper. The data acquisition software's primary task is to initiate and control collection of data from all elements of the detector array. This includes collecting data from the DOMs, processing that data stream through one or more triggering algorithms and assembling those triggered hits and sensor data into an event stream. The data acquisition process is divided into "runs" of several hours and, during normal data-taking operations, the system cycles through sequential runs indefinitely. The event stream output is handed off, at run-time, to the Processing and Filtering component (PnF) where it is written to tape with a filtered subset put on the satellite queue for transport to the northern hemisphere for further off-line analysis.

The core DAQ architecture (see Figure 1) consists of 4 levels: components, detector configuration, component control and run control.

At the component level, the StringHub controls a single string, including the initialization and state management of the DOMs for that string, collection and time-sorting data from the DOMs and in the event of a DOM becoming unresponsive, dropping individual DOMs from the current run. As such, there are as many StringHub components running as there are Strings in the current run's configuration. The trigger components read the time-calibrated data from all the strings and other sources and, using one or more triggering algorithms, produce descriptions of events which span the detector. The EventBuilder, based on the event description from the trigger then collects all the data from the StringHubs, and produces an event stream. The

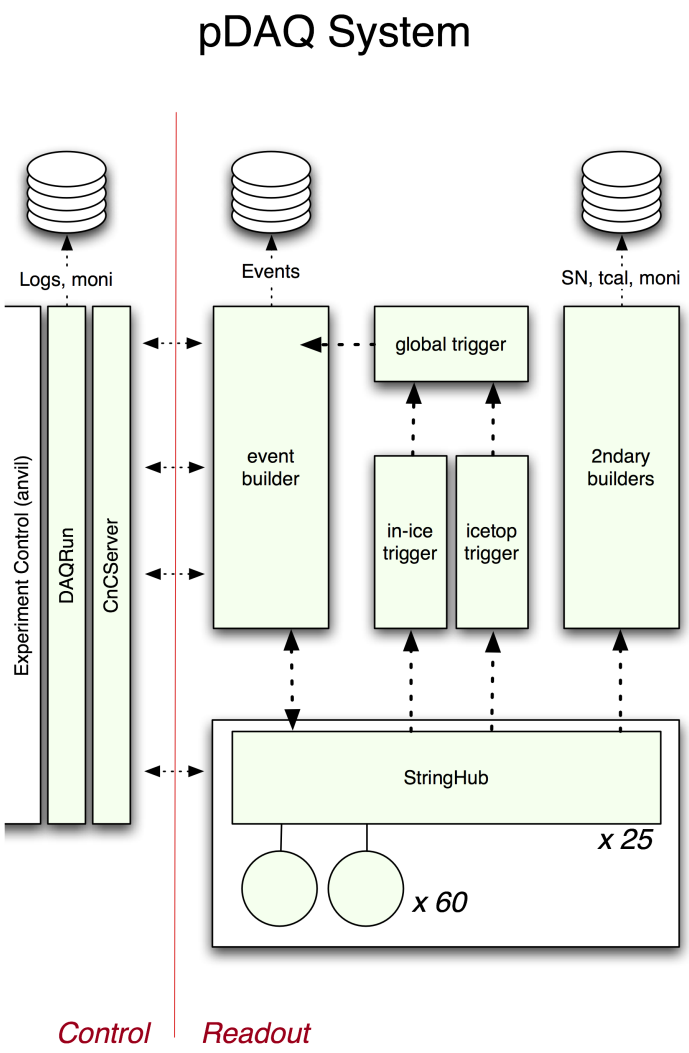

Figure 1. Current high-level pDAQ Architecture: on the right, data flows from the bottom up, control components are on the left.

Secondary Builders components collect 3 "secondary" streams of data from each DOM, as harvested by the StringHubs: time calibration data (used for accurate path reconstruction); monitoring data reflecting the state of each DOM; and PMT scalar readouts which form the basis of supernova detection for IceCube.

Detector configuration stores the various sensor, software and compute cluster configurations, referenced at both deployment time and run initialization. These identify the number of strings to be included in a given run set, the configuration parameters for each individual DOM, the trigger algorithms and the distribution of which components are running on which physical machines in the compute cluster. This flexibility allows for sets of cluster configurations for running simulations or against real DOMs on our test cluster in the North.

The component control server (CnCServer) manages the individual components and packages them into run sets which are manipulated at the run control level. To build a run set, CnCServer connects the input and output streams of the StringHubs, trigger and builders and starts, configures and stops the components according to instructions from the run controller.

The run controller (DAQRun) provides an abstract interface of the DAQ to Experiment Control. (Experiment Control, not part of the DAQ, also controls other subsystems in the detector in addition to the IceCube DAQ.) DAQRun uses CnCServer to control the DAQ, but can also kill and restart components at any of the lower levels. This level could conceivably be combined with the component control level, but has been kept separate in case there is a need to someday partition the instrument into separate pieces in order to do verification or calibration runs in one section while data-taking runs continue using the rest of the instrument. If that were to happen, there would be multiple instances of DAQRun talking to a single CnCServer instance.

2. Experiences & Observations

2.1. Distributed, Multi-Institutional Collaboration

The IceCube project is a large one, with collaborators from all around the world. Originally, the DAQ development team had participants from both the United States and Europe. However, by the time of the first string deployment at the South Pole, the software development group consisted solely of contributors working at 4 locations (and 3 time zones) within the US.

We scheduled frequent, mandatory phone conferences at times that would allow maximum attendance. At roughly three month intervals, we held face-to-face meetings at participating institutions. Furthermore, all team members shared cell phone numbers, subscribed to dedicated development discussion email lists, common IM chat rooms, etc. All these tools enabled communication to take place as easily and often as necessary. As a result, the development team developed a remarkable level of integration and cohesion which would prove to be critical during both integration & system testing as well as during deployment.

It is worth noting that, while the project would have benefited from having the entire implementation team co-located for the duration of system design and implementation, this was not deemed practical. It proved equally effective to promote, among team members, regular and frequent communications and regular face-to-face meetings. In other words, frequent (i.e. daily) communication mitigates many of the problems stemming from geographic dispersion. This level of communication proved problematic for collaborators working at large time deltas from central US, meaning that those who could not easily and frequently communicate (e.g. those in European time zones) moved on to other IceCube software activities.

Beyond the added communication complexity inherent in a distributed collaboration, the impact of working within a multi-institutional collaboration also had an effect on the software development process. Having multiple institutions in a project introduces a project structure where a separation frequently exists between project hierarchical management and individual institutional supervision. This separation significantly increases the likelihood that a strong project management approach will not be realistically possible and a more consensus based strategy will need to be employed at times. This doesn't mean that it is impossible for strong leadership, just that where project management crosses institutional boundaries, this creates a situation where authority and responsibility are not tied together. We've seen that having flexibility on when and how to take a top-down vs. a bottom-up management approach to be very effective depending on the particulars of the situation at hand and the people involved.

For example, a task which exists entirely within a single institution where contributors work side-by-side in daily contact can work in a more traditional top-down management structure - e.g. working through the interface, implementation and testing details of a specific sub-component. Whereas tasks requiring coordination of multiple teams across several institutions with the with less daily contact and greater potential for differences of opinion on requirements, design details or even tool selection, will benefit from a bottom-up consensus-based approach.

2.2. Project Management

Software development in industry, generally, has a very top-down management structure where responsibility is directly tied to authority. In research, as described above, this may not always be possible and in those situations, calls for a different project management approach. In our experience there are two primary reasons for this separation: projects can include individuals from multiple institutions on a single development task and the collaboration consists of experts from multiple disciplinary backgrounds with different views and interpretations of system requirements. These two reasons are related in that they are born out of the inherent size and complexity of an undertaking like the IceCube project. But given that funding agencies often encourage (and at times require) institutional diversity in smaller projects, insight on how to navigate these (at times very challenging) collaborative waters, we hope will additionally prove helpful in smaller projects.

One dramatic experience drawn from our work here is the realization of how differently people from different disciplines approach the development of software. For example, those from a hardware engineering background have a different approach than software engineers, clearly due to the difference in materials. Software, being infinitely malleable unlike hardware, allows a greater degree of freedom in design and revisions. Physicists, who generally learn to program as a means to another end (producing meaningful physics results) approach software engineering differently than those with a formal computer science background.

The combination of these varied (and at times fiercely defended) approaches towards the relative importance of software implementation design and detector performance created a project management challenge. Considerable flexibility is required on how to proceed, based on the nature of the specific task, people and institutions involved. This realistic and honest assessment of what is achievable, despite "*widely understood best practices*" and how it might be possible to achieve it, within a given project is one of the central lessons learned from working on the IceCube DAQ.

A successful example of a bottom-up managerial approach was the creation of a regular "Change Control Board" (CCB) meeting. During this weekly phone call we culled through the current queue in our issue tracking system, organizing both features and bug-fixes into releases of the DAQ. These could range from simple milestone snapshots through fully certified DAQ system releases for deployment to the South Pole. These conference calls, though at times tedious - with detailed discussions of issue after issue, were extremely beneficial in establishing an overall development structure and developing a solid group understanding of how future development would proceed. A consensus based approach was taken in guiding the CCB meetings, with detailed design discussions discouraged in favor of deciding on *what was next*. For some issues deadlines were very important and those releases followed a more date-driven schedule. Although in practice the majority of releases were feature driven with each weekly release milestone picking up for integration and system testing the issues resolved from the previous week. An additional benefit was the popular adoption of the Mantis issue tracking system (despite some initial preferences expressed for Bugzilla).

2.3. Team Management

One challenge faced by the team was how to assign tasks and responsibilities. It is often desirable to partition a large effort into well defined chunks which are assigned to specific individuals. But software developers vary greatly in their skills and weaknesses, which are not always apparent from the outset. This fact, combined with the fact that the components varied in complexity, meant that some components took quite a bit longer than others to finish, thereby holding up the whole effort. A more fluid approach was used later in the project where developers readily crossed boundaries between components (a necessity due in part to shrinking manpower), leveraging that person's abilities (and availability) wherever appropriate. This was also made

more tenable because of the established integration and unit test frameworks, which reduced the risk of inadvertently breaking something due to a contributor's relative unfamiliarity with a piece of code.

A contributing factor to the differing times taken to develop the various components was the use of graduate students for software development. This ultimately became problematic not because of a lack of ability or skill on the part of the students but due to the part-time and temporary nature of their participation. In some situations the use of students for development worked out well but those were situations where they were supervised by a full-time participant on the project.

2.4. Collaborative and Development Tools

The most effective way we've found to navigate the complexity of a project structure like this is through effective communication. In a sense, each contributor needs to over-communicate to compensate for the "loss" which is inherent in distributed collaborations. We employed many of the now common collaboration tools to this effect. These ranged in their technical sophistication from direct telephone calls, regular weekly conference calls, email lists, instant message chats and chat rooms, to VoIP applications such as Skype. Periodically we would travel for meetings where we could meet face-to-face for discussions and working sessions. These proved to be essential in establishing good working relationships supported subsequently via the above tools.

In addition to these, application development tools also serve as communication tools. A version control system and a bug tracking system with good logging features and email or RSS feed notifications, goes a long way towards keeping developers abreast of current development activities. Initially we were using the set of tools described by Patton and Glowacki [2]. As developer preferences and other tools emerged this set of tools were replaced in preference to a combination of Subversion (svn) for version control combined, Mantis for issue tracking & simple project management and Maven as a build, unit test framework, dependency tracking and release assembly tool.

One lesson learned in using in-house development tools is the importance of understanding the effort required in supporting these tools. This support takes two forms: fixing bugs & adding required features and, of great importance in a large distributed collaboration, having an expert available as an advocate and instructor for the selected tool set. Inevitably there will be times when a tool is not serving the needs of an individual or group of users. This can be because the tool simply lacks the functionality desired, the functionality exists but in a way undiscovered by the users, or users are trying to accomplish something counter to the development philosophy upon which the tool is intended. If the tool simply lacks the functionality, is broken in some way or requires updated documentation, having reasonable response from the author is needed. If the users are trying to apply the tool in a way it was not designed to be used, then a discussion needs to take place.

The presence of this tool advocate (which does not have to be the author) to document, educate and at times mitigate on the proper use of the tool and what can be expected of it, is necessary. In the absence of this advocate adoption of the tool will either evaporate as users will move to use their tools of preference or use of the tool will become contorted and confusing when the underlying concepts on the use of the tool have been lost. Both of these situations can then lead to a fracturing of tool sets used within the project, which in itself increases complexity in managing the project.

2.5. The Importance of Communication

When working within a group, communication is of paramount importance. This is especially true when working remotely from others as the possibility for misunderstandings, or assumptions to creep in increases with the number of remote parties. Requirement specification is a form

of this communication, and so are all the collaborative tools mentioned above. In short, our experience is that it is difficult to over-communicate in distributed efforts such as this.

An interesting example of this is that some of our best work was done during the austral summers at the South Pole, when many of us spent more time working together in the same room than at other times of the year. The austral summer at the pole is the only time when the actual drilling and construction of the detector can take place, and hence when not part of the physical deployment crew, we would have unique access to actual strings in the ice, if not the full detector, for scale testing of the DAQ. These, often marathon working sessions, proved to be essential in solving scalability issues which are impossible to simulate with our test systems - especially when the detector is in the process of increasing in size. Sitting in the same room, working closely together proved to be very productive.

2.6. The Role of Requirements Specification

Nearly all development methodologies stress the importance of specifying requirements. Because software revisioning is so easy, as compared to hardware development or experimental science, software development can easily be seen as an infinitely mutable process and therefore has a corresponding effect on the effort put into specifying requirements.

The DAQ system requirements were derived from early statements of overall detector performance goals. Descriptive requirements of the detector's physics performance were developed as part of the initial design documentation and were refined through a series of pre-construction project reviews. As the project entered the construction phase, individual engineering groups, both hardware and software, started developing requirements to govern design and performance of their particular subsystem. While some project subsystems carried this process through to the lowest layers of design, there were inconsistencies across project subsystems.

In retrospect, more attention should have been given to the very different ways in which physics and engineering requirements are expressed and interpreted. While the former are primarily descriptive in nature, engineering requirements tend to be expressed more formally and often prescribe testing techniques and organizational structures for allowing exceptions (e.g. change control boards). Within the project, there was, and continues to be, a clear and pervasive understanding of overall system performance goals. But, when resolving individual subsystem implementation issues or weighing required features against slipping implementation schedules, it was difficult to navigate between the overall descriptive project goals and the proscriptive subsystem requirements with their formal change control processes.

A more flexible project management approach may have avoided the situation where decreased importance was placed on specifying formal requirements to help resolve everyday implementation issues. As a result, this had a substantial effect on the effective software development methodology which was used.

2.7. Development Methodologies

Initially an Agile[3] or Extreme approach was taken, which in a nutshell, starts with the assumption that system requirements are clearly understood; but, as time passes, even during the implementation process itself, these requirements will change and evolve. These changes are managed by advocating rapid, small well-scoped development cycles, between which requirements are re-evaluated. Support for this in the development tool set is vital, for example having an effective unit-test framework easily running regression tests. And, given the rate at which implementation must respond to changing requirements, effective code release management and discipline are essential. Early in the project, we assembled a tool set that supported this development methodology and it was applied on an individual, or small working-session group level. As a result of the changing role of formal requirements within the project,

the task of creating requirements at the lower levels was never fully realized. This, in turn, had an effect on our development practices.

What materialized in its place was a method of deploying, typically into a developer's sandbox and on our test cluster, software implementing a particular set of functionality for integration testing with the current DAQ baseline. Working up from there, more full system testing evolved and higher-level integration issues with other IceCube software components (both "above" and "below" the DAQ) were addressed. This somewhat ad-hoc development method often exposed differences in participant's understanding of the architecture, which led to discussions and at times difficult decisions, but generally working code won over unimplemented designs.

2.8. Testing

Unit testing is another method taken from the toolbox of Agile methodologies. The idea is that as code for the project is developed, a suite of automated tests is written in parallel by the programmer to verify that the code is working correctly. These tests can then be run after each change to the code, to ensure that no functionality has been changed. When a change causes an error in the test suite, both the test code and the change are examined to determine if the problem was caused by the change or if the test suite was in error, and the appropriate fix is made.

Building up a set of unit tests which exercise the code, while initially difficult, becomes essential in a project with a long development period. Broad changes across many code modules or changes deep within some vital algorithm – changes which might normally be avoided because of the likely hood that things would break – can be made quickly and with assurance because the unit tests can verify with a high degree of certainty that nothing unexpected has changed.

A separate but crucial class of tests consists of integration testing, where all the components are brought together. For the IceCube DAQ, a test stand located in Wisconsin, consisting of ten or so nodes, was created, attached to a small number of sensors. This allowed for a basic sanity check of the system as a whole. The additional ability to run simulated DOMs allowed a user to run the system on a standalone node (such as a laptop), or on the test system simulating 40 or even 80 strings (in this case, multiple StringHub instances were deployed per physical processor). While still lacking in the ultimate realism of the production system, the simulation was used to study the scaling properties of the system before the complete production array was deployed.

Unfortunately, there were not adequate resources to duplicate the entire production system in the North, so there have often been surprises when deploying new software to South Pole. This makes release management and the ability to quickly roll back to previous releases all the more crucial.

2.9. Off-the-Shelf vs. "Roll your Own"

A fundamental question which arises repeatedly in software development is whether to reinvent something or to re-use an existing infrastructure. There are pros and cons on either side. Using an existing system may be employed quickly after an educational investment and potentially modifying your design to fit the model assumed by the system. It may also be required to invest developer time to customize or extend an existing system. Rolling your own solution has the advantage that it will be exactly what you need with familiarity and expertise on the system in-house. This though comes at a potential cost if documentation is insufficient and there is turn-over in your personnel. Unfortunately it is often difficult to determine the costs and benefits associated with a given path in advance or even in hindsight.

Our example here is how JBoss was initially chosen as the application and clustering framework for all the DAQ components, as well as for other IceCube software components. As development matured the selection of JBoss and our specific use of it was called into question.

While the original proponents attempted to answer these questions, it became clear that the trend of the group was against continuing the use of JBoss. Ultimately the functionality we used it for was replaced by significantly more light-weight components of our own.

This was a very contentious decision within the group. In fact, the merit of this decision (though now in the past) continues to be a point of disagreement between some of the authors of this paper. On one side the argument is made that the newer system is much more lightweight, faster, easier to understand & modify and therefore was worth the investment in the course correction. On the other side, the point is made that expertise has just moved from one group to the authors of the new system and the the same reduced complexity, scaling & performance improvements could have been achieved through a similar or even smaller re-factoring investment.

What can be drawn from this experience are at least two points covered in general above: the importance of near constant communication, and advocacy for and against decisions made by the group. Similar to the need for an advocate for the use of in-house development tools, an advocate for design and technology selection decisions can help a group confront conflicts in opinions earlier and potentially save the group significant effort and time.

2.10. Benefits of Rapid Prototyping and Rewriting

A basic principle of Agile Development is to favor working code over written requirements and specifications. This implies that as requirements change and ones understanding of the problem improves, code must be quickly adapted or rewritten to match the shifting domain. The rewrite of the control architecture described above took place initially as a rapid prototyping effort by two of the developers over a few days. After the prototype was seen to work successfully, it was simply expanded to cover all the required use cases. The use of Python for this relatively high-level problem domain made the development very fast. Other important DAQ components were rewritten and a significant reduction in the size of the code base was achieved. Though rewriting takes time, it often allows for a much cleaner and more maintainable implementation, as the developers learn from previous efforts.

3. Conclusions

Ultimately, the outcome of this effort has been a successfully working DAQ, which is the "proof of the pudding". What we can draw from our experiences and observations here is that at times, technical considerations need to take a back seat to the realities of the project and it's participants.

Communication is a top priority for development in distributed teams like this. Any and all communications tools should be employed with developers receiving constant encouragement to evaluate and focus on those tools that are most effective. Face to face meetings are irreplaceable in creating the rapport needed for effective development when most time is spent working remotely.

Finally, management style, tool selection & ownership, and implementation decisions must take into account both the individuals and the institutions involved and how they will be most effective in helping the project achieve its goals, despite what might be considered in other contexts a more correct approach.

Acknowledgments

This work was supported by National Science Foundation - Physics Division and by the Office of Science, Office of Basic Energy Sciences, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] A. Achterberg et. al., "First year performance of the IceCube neutrino telescope", *Astropart.Phys. 26 (2006) 155-173.* [doi:10.1016/j.astropartphys.2006.06.007].

[2] S. Patton and D. Glowacki, "IceCube's Development Environment", *In the Proc. of 2003 Conf. for Computing in High-Energy and Nuclear Physics (CHEP 03), La Jolla, California, 24-28 Mar 2003, pp MONT001* [arXiv:cs/0306057].

[3] K. Beck et. al., "Manifesto for Agile Software Development" http://agilemanifesto.org/